FINAL REPORT:
EXTENDING LEARNING IN SOAR

John Laird
University of Michigan
2260 Hayward
Ann Arbor, MI
48109-2121
Phone: 734 647-1761
Fax: 734 763-1260
laird@umich.edu

March 14, 2006

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	03.14.2006	FINAL 6/29/2004-9/30/2005

4. TITLE AND SUBTITLE
Extended Learning in SOAR (FINAL REPORT)

5. FUNDING NUMBERS
Grant: HR0011-04-1-0044

6. AUTHOR(S)
John E. Laird

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
The Regents of The Univ. of Michigan
Division of Research Dev. & Admin.
3000 South State Street
Ann Arbor, Michigan 48109

8. PERFORMING ORGANIZATION REPORT NUMBER
F010725 - 048136

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Defense Advanced Res. Projects Agency
Contracts Management Office
3701 North Fairfax Drive
Arlington, VA 22203-1714

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for Public Release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 Words)
The major goal of this project was to develop the science and technology for building autonomous knowledge-rich *learning* agents – computational systems that have significant competence for performing tasks without human intervention, but also have the ability to learn new tasks and improve almost any aspect of their behavior through learning. Our plan was to concentrate on building agents with a variety of architectural learning mechanisms, including reinforcement (to capture statistical regularities), episodic (to capture experiences) and semantic learning (to capture facts).We also planned to study their integration, including integration with Soar's chunking mechanism (which captures procedural knowledge).

The result of this project has been to develop initial versions of episodic and semantic memory, while at the same time creating a robust implementation of reinforcement learning in Soar. By the end of this project, we had implementations of all three learning mechanisms, and each of them integrated with Soar's chunking mechanism. In follow on work, we are refining episodic and semantic memory and creating a complete integration of these learning mechanisms.

14. SUBJECT TERMS
Cognitive Architecture, Machine Learning, Artificial Intelligence

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	

INTRODUCTION

The major goal of this project was to develop the science and technology for building autonomous knowledge-rich *learning* agents – computational systems that have significant competence for performing tasks without human intervention, but also have the ability to learn new tasks and improve almost any aspect of their behavior through learning. Achieving this goal requires identifying the types of knowledge required by complex human-level agents, as well as research into mechanisms for learning those types of knowledge. More broadly we want to identify the learning mechanisms that are needed to create more robust, intelligent agents that can pursue multiple goals and survive in environments where novel, unexpected problems arise. Our plan was to concentrate on building agents with a variety of architectural learning mechanisms, including reinforcement (to capture statistical regularities), episodic (to capture experiences) and semantic learning (to capture facts).We also planned to study their integration, including integration with Soar's chunking mechanism (which captures procedural knowledge).

The result of this project has been to develop initial versions of episodic and semantic memory, while at the same time creating a robust implementation of reinforcement learning in Soar. By the end of this project, we had implementations of all three learning mechanisms, and each of them integrated with Soar's chunking mechanism. In follow on work, we are refining episodic and semantic memory and creating a complete integration of these learning mechanisms as show in Figure 1.

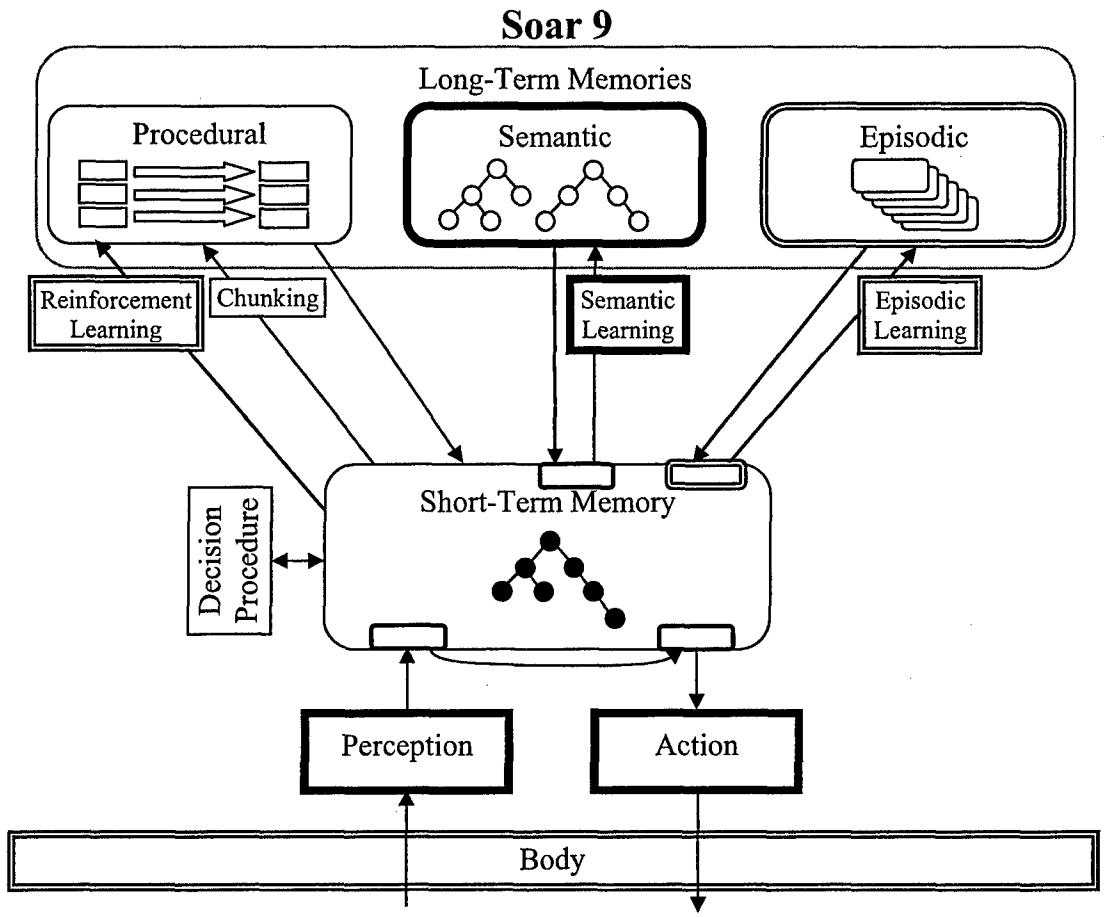

Figure 1: Structure of Extensions to Soar

Details of each of the learning mechanisms is described below. The integration of these learning mechanisms with chunking (Soar's procedural learning mechanism which compiles problem solving into rules) required a significant extension to chunking. Before adding these learning mechanisms, Soar's reasoning was mostly deterministic. As preparation for reinforcement learning we extended Soar's decision procedure to fully support non-deterministic decision making. This violated an assumption of chunking (and most logic-based reasoning systems), namely that if a line of reasoning was pursued once and achieved a result, that line of reasoning could be compiled into a rule. However, when the decisions are made probabilistically, compiling a line of reasoning can solidify non-optimal behavior very early on. This led us to revise chunking so it is more conservative, compiling reasoning only when the confidence in each decision is very high. Retrievals from episodic and semantic memory can also be probabilistic (they provide "best" matches), so this extension to chunking also including the requirement that retrievals from those memories must be made with very high confidence (actually, there must be an exact match).

The other significant result of this research was that it led directly to the creation of a successful proposal to the DARPA Biologically-Inspired Cognitive Architecture program. Our BICA project picked up directly from the results of this project.

EPISODIC LEARNING

Episodic learning captures a history of what happened to the entity. This raw information is not directly useful for decision making because it is retrospective not prescriptive. However, by recalling relevant memories about the effects of actions in the world and their impact on the goals and desires of the entity, the entity can use episodic memory to improve future decisions. Of course, recalling and reasoning about episodic memories is a reflective activity that takes time and so the costs of reflective reasoning using episodic memories can be prohibitive in time critical situations. Nevertheless, an autonomous entity is not always under time pressure. It will have many occasions when it can reflect back on prior activities, analyze what went right or wrong, generalize across multiple experiences, and learn knowledge that will influence future decisions.

Episodic memory has several characteristics that may have an impact on its implementation. Below are some of the most distinguishing in contrast to other types of memory.

Autonoetic:	The retrieved memory is distinguished from current sensing.
Autobiographical:	The rememberer remembers the episode from his or her own perspective.
Variable Duration:	The time period spanned by a memory is not fixed.
Temporally Indexed:	The rememberer has a sense of the time when the episode occurred.

One of the results of our research was to identify the uses of Episodic Memory. This process began as an analysis of environments where episodic memory would be effective. It has evolved into a catalog (below) of the cognitive capabilities that are supported by episodic memory. When an agent is ineffective due to a lack of episodic memory, it is because it lacks one or more of these cognitive capabilities when it is sensing, making a decision or learning:

- **Noticing Significant Input:** Episodic memory provides a measure of the familiarity of a set of features of the environment within a familiar context. Conversely it allows you to recognize when an action does not produce an expected outcome.
- **Virtual Sensing:** Events or sensing that may not have been relevant to the task when experienced may unexpectedly become relevant in the future (e.g., Where did I last see my car

keys?). Episodic memory provides an avenue for retrieving past sensing. In effect, it provides another sensory input to the decision process.

- **Action Modeling:** Episodic memory allows an agent to predict how its environment will change as the result of a given action by recalling episodes where it experienced a similar situation.
- **Retroactive Learning:** Episodic memory allows previous experiences to be relived or rehearsed once the resources are available so it can be reanalyzed with new knowledge or experience. **Explaining Behavior:** The ability to remember what you did and why you did it allows you to explain your actions to others and allow them to instruct you or you to instruct them (e.g., Why did you go left instead of right?).

Another thrust of our research was to determine a set of requirements for an episodic memory system. They are summarized below:

- **Task Independent:** A task independent episodic memory can be used in domains that the agent encounters but is not programmed for.
- **Low Resource Demand:** If the episodic memory system interrupts, or severely slows the agent's reasoning, then the cost of the system can outweigh its value.
- **Non-Invasive Integration:** An episodic memory system that requires changes to the agent will be less useful to the agent's developers.
- **Supports Both Deliberate and Spontaneous Retrieval:** The most obvious form of episodic retrieval is deliberate retrieval wherein the agent specifically attempts to remember a particular episode that matches a given cue. However, an agent can also benefit from spontaneous retrieval that occurs when a particular episode is strikingly similar to something from the past.
- **Negative Cue:** The ability to specify features that should *not* be in a retrieved memory seems essential to some tasks. (e.g., "When was the last time we *didn't* go to your parents' house for the Thanksgiving?")
- **Supports Recursive Retrieval:** An agent that monitors its behavior must not only be capable of remembering an episode but should also be able to remember remembering an episode.
- **Supports Sequences of Episodes:** Once an agent has retrieved an episode, it should be able to remember what happened next. Without this feature, the episodic memory is unable to provide the agent with the action modeling and retroactive learning cognitive capabilities.
- **Episodes Convey a Sense of Time:** Knowing the order in which past events occurred can be critical to an agent's success.
- **Strength of Match:** Information about how strong the match is between a retrieved memory and its cue can be useful to the agent because it allows the agent to adjust its confidence in any decisions it makes because of the content of the memory.

We have many difficult decisions to make in trying to create a computational implementation of episodic memory. Episodic learning can be decomposed into the following major phases:

- encoding – how a memory is captured and stored;
- storage – how a memory is maintained;
- retrieval – how a memory is retrieved; and
- use – how the memory will be used by the agent.

Some of the most basic design decisions, such as the structure of a memory, have impact across all phases. The original framework was based upon the work of Endel Tulving (1983) and then refined as we encountered issues or other research. There is no guarantee that this is a definitive framework, but it does encompass all implementations and ideas we have considered to date.

- **Encoding**

- **Encoding initiation:** when an episode is recorded. Initiation is automatic, but there are many possible events that could trigger encoding. This might be as simple as recording a new memory at regular intervals. Alternatively, a new episode might be recorded whenever the agent takes an action in the world, whenever there is a significant change in its sensing or when something unexpected occurs.
- **Episode determination:** what information is stored in an episode. An AI agent's state consists of a set features that represent its current sensing as well as a set internal features that are derived from its processing. The content of this episode consists of a subset of these features. This subset could consist of only the sensing information or only the internally derived features. The episode might consist of only those features that have been attended to by the agent. It is at this stage that the episodic memory system must select what is "important" about an episode.
- **Feature selection:** which features in an episode will be available for match during the retrieval phase. In effect, this determines which parts of the episode will be matched to a memory cue during retrieval. The episode's features may include the entire episode content or as little as one or two key elements. If a subset is selected, this reflects the episodic memory system's prediction about what features will be needed to match future memory cues.

- **Storage**
- **Episode structure:** how the encoded episode is stored. The structure of the episode is often dependent upon the architecture upon which it is built. The structure must support the other phases of the memory system.
- **Episode dynamics:** how stored episodes change over time. This may manifest as cross indexing with other memories. It may include a form of memory decay such as a loss of detail, a decline in activation, a merging with other memories or outright removal from the episodic store.

- **Retrieval**
- **Retrieval initiation:** how memory retrieval is triggered. Often the agent will initiate memory retrieval in order to accomplish a specific subtask where it deems that previous experience may be useful. Episodic retrieval may also be triggered automatically by the presence of keenly familiar elements or unexpected sensing.
- **Cue determination:** determining which data are used to cue the retrieval of an episode. Memory cues may consist of a partial memory that can be matched directly to other memories in the store. They may also consist of a few key features. Finally, a memory cue may consist of a command relative to the previous retrieval (e.g., "What happened next?").
- **Retrieval:** selecting which episode is retrieved given the current cue. This is the critical matching phase of memory retrieval. Matching can be exact or partial. Different features of the cue can be given variable weight in the match. Multiple memories may be retrieved or just a single match. The match algorithm can also be influenced by the agent's current sensing and internal state.
- **Retrieved episode representation:** what aspects of the episode(s) are retrieved and how they are represented. The result of a match can be as simple as a Boolean yes/no (i.e., recognition). Typically, the episodes are recreated or referenced in some location where the agent can examine them.
- **Retrieval meta-data:** what meta-information about the retrieved episode is available. Possible meta-data that could be made available includes: information about the strength of the match between the episode and the cue; temporal information describing when the episode occurred and information about when it has been retrieved in the past.
- **Use**

- Once the episode is retrieved, how it is used to aid reasoning? This is not a part of the design of the episodic learning system, but depends on the capabilities of the embedding architecture, general methods, and task knowledge. We can expect episodic memory to be used for one or more of the cognitive capabilities listed in the previous section.

To date, we have created a working episodic memory system for Soar called Soar-EM. As part of our iterative methodology, Soar-EM has undergone two large reimplementations and several smaller modifications as we have learned more about episodic memory and its use by intelligent agents. The current implementation uses a separate episodic memory, with episode that consist of a subset of working memory that is selected in a task independent manner. Retrieval of the episodes is modeled after the use of buffers in ACT-R, where a retrieval is made when a cue is placed in a special location in working memory. The current implementation's external episodic memory store supports the use of a partial matching algorithms. To allow the agent to retrieve multiple episodes, we added support for a direct command to retrieve the next episode in temporal order (rather than attempt to retrieve it via a query).

The current episodic memory system fits within our episodic memory framework as follows:

Encoding

- **Encoding initiation:** A new memory is encoded each time the agent takes an action in the world. We have also experimented with recording new episodes whenever there is a significant change in working memory.
- **Episode determination:** The content of an episode consists of a large portion of the agent's working memory including its input (sensing) and output (actions in the world). This subset consists of all elements of working memory whose activation level has not decreased to the point where the element would be removed in a strict psychological model.[1]
- **Feature selection:** All features in the episode can participate in retrieval.

Storage

- **Episode structure:** We have experimented with two different episodic memory structures in the current implementation. Both structures use a single data structure (called the working memory tree) that holds a single instance of all elements that have ever been in working memory throughout the agent's existence. In the first approach, episodes are stored in an instance-based format. Each episode consists of a list of pointers to each of the working memory elements it contains, in a canonical order. In addition, all elements in the tree have pointers to the memories that contain them. The activation level of each element at the time of storage is also recorded in each episode. This structure is illustrated in Figure 1.

  The second approach is to store the episodes implicitly in the working memory tree via a series of time increments (ranges of decision cycles) on each node in the tree. This makes the required storage linear in the number of changes to working memory instead of the number of elements in working memory. The ranges indicate the cycles when the associated working memory element was in the agent's working memory. This approach precludes the ability to efficiently store the activation level of each element in a given memory but it uses significantly less space (see section **Error! Reference source not found.**). This structure is illustrated in Figure 2.
- **Episode dynamics:** Episodic memories still do not change over time.

[1] Working memory activation was originally added to Soar to support memory decay models. Thus, when a particular working memory element dropped below an activation threshold it was removed from working memory.

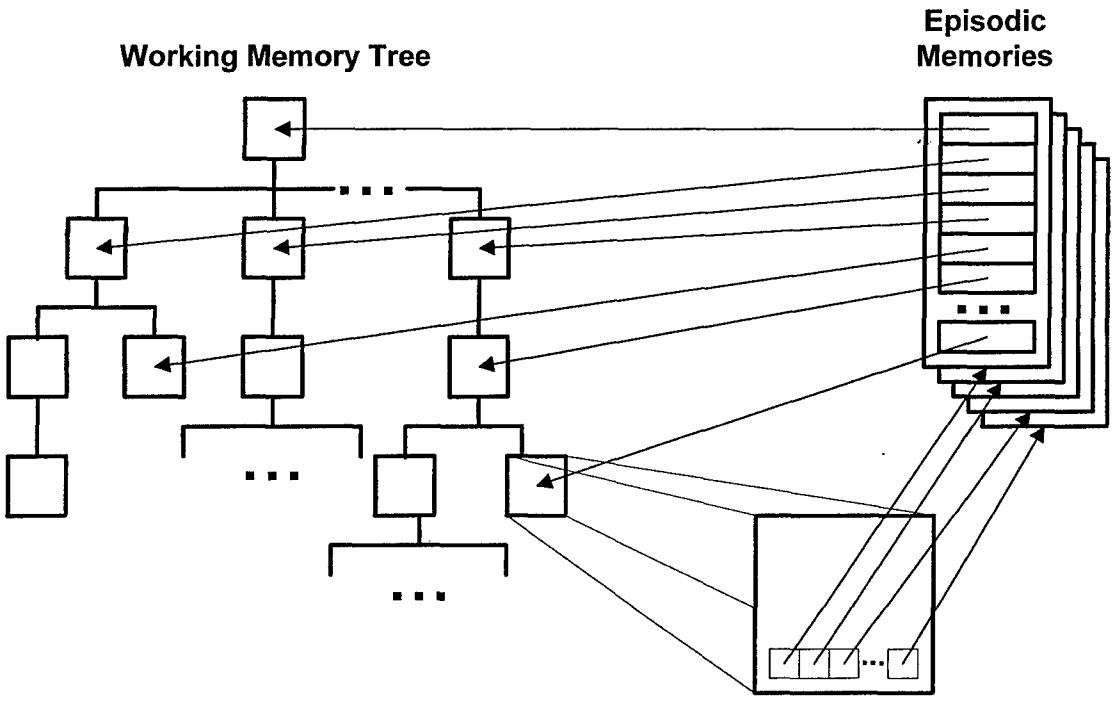

Figure 1: Soar-EM: Instance-Based Approach

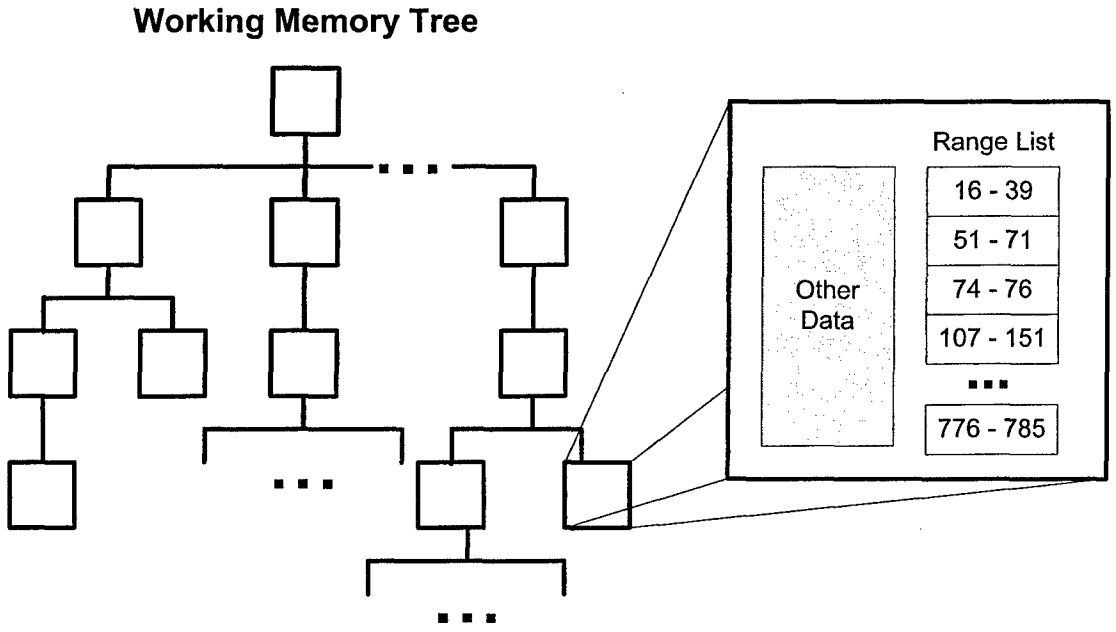

Figure 2: Soar-EM: Interval-Based Approach

Retrieval

- **Retrieval initiation:** Retrieval is initiated deliberately by the agent when the agent creates a cue. There is currently no support for spontaneous retrieval.
- **Cue determination:** A cue is deliberately constructed by the agent (using rules) in a reserved location in working memory. The cue can have any number of working memory elements.
- **Retrieval:** During episodic retrieval, the cue is compared to all stored episodes in order to select the episode that "best matches" the cue. The match is determined by totaling the number of working memory elements that are shared between the cue and the episode. In the instance-based approach, the match is weighted by the activation level of the WMEs when they were recorded into the episodic memory. In the interval-based approach, the match is weighted by the activation level of the WMEs in the cue. Once an episode has been retrieved, the agent can also retrieve the next episode in temporal order via a special "next episode" command.
- **Retrieved episode representation:** The complete episode is retrieved in a labeled area of working memory to avoid confusion with the current state of the agent.
- **Retrieval meta-data:** Currently there is no meta-data retrieved with the episode.

Instance-Based vs. Interval-Based Retrieval Algorithm

This section provides a more detailed comparison of the two approaches we've implemented for storing episodic memories.

Instance-Based Approach:

The instance-based approach was implemented first and its algorithm is as follows (refer again to Figure 1 above):

1. The system simultaneously traverses the cue and the working memory tree. For each entry in the cue, the corresponding entry in the working memory tree is located.
2. Each element that is matched in the tree contains a list of references to every episode that contains it. A list of all episodic memories that contain at least one element from the cue is created by merging the references from each matched part of the cue.
3. The complete cue is then compared to each episodic memory in the newly created episodic memory list and the one that best matches the cue is selected.

This algorithm requires O(nm) comparisons to find the best match to a given cue (where 'n' is the number of elements in the cue and 'm' is the number of episodes that elements of the cue appear in). Note that the size of the cue will be much smaller than the number of episodes and not grow over time. While the cardinality of 'm' could equal the size of the entire episodic store, in practice it is much smaller because a given entry in the cue is unlikely to appear in all the stored episodes. Moreover, the size of the cue limits the number of memories that need to be directly compared to the cue. In addition, if the agent is moving from tasks to task, so that the number of episodes with common features does not grow, the time required will be much smaller than O(nm) might imply. Our results do show growth, but in a single task where all of the episodes share common features (this is the worst case in terms of performance for the algorithm).

Interval-Based Approach:

The interval-based retrieval algorithm follows these steps:

1. The system simultaneously traverses the cue and the working memory tree. For each entry in the cue, the corresponding entry in the working memory tree is located.
2. Each element that is matched in the tree contains a list of ranges. Each of these lists is set aside and each range in the list is assigned a match score equal to the activation level of the associated cue element.

3. All of the selected lists are merged together into a single list of ranges. If two ranges partially overlap, they are split into two or more separate ranges. For example, if one list contains two ranges (1-10, 15-20) and the other list contains one range (8-18) the merged list will contain six ranges (1-7,8-10,11-14,15-18,19-20). Each range in the merged list has a match score equal to the sum of the activation levels of all the ranges that entirely covered that range.
4. The merged list is traversed to locate the range with the highest match score (activation level). The number(s) in that range represent the cycles in which the best matching episode was recorded. (In the event of a tie, the most recent cycle is selected.)
5. The episode can be recreated by traversing the working memory tree and creating working memory elements for each node that contains the selected cycle in one of its ranges.

The complexity of this algorithm is O(nr) where 'n' is the number of items in the cue and 'r' is the total number of ranges that must be examined. This complexity can also be expressed as $O(n^2l)$ where 'l' is the average number of ranges in each list that is merged in the matching process. As before, these two variables are not independent in practice. At worst, we expect the growth to be linear in the number of episodes because the size of the cue is relatively constant as it was in our test domain. As in the prior algorithm, the growth will be minimized if the same features in the environment are not continually encountered (so that r is small). However, this algorithm has the additional advantage in that it is sensitive to only changes in features, so that the growth could be significantly less than the instance-based algorithm if features change values slowly.

The most expensive processing required by the matching algorithm is the merging process. This merging requires that all the ranges in all the lists be sorted twice: once by their high numbers and then again by their low numbers. It may be possible to reduce this time by using a faster sorting algorithm.

RESULTS FROM THE CURRENT IMPLEMENTATION

The domain we selected for our experiments is called Eaters. An "eater" is a Pacman-like agent that moves around a 16x16 grid world (see Figure 4). Each cell in the grid is either empty or contains a wall, normal food (● = 5pts), or bonus food (■ = 10 pts). The eater is able to move in each of the four cardinal directions unless there is a wall in its way. Each time it moves into a cell containing food; it eats the food (receiving the appropriate score). When an eater leaves a cell it becomes an empty cell. The eater's goal is to get the highest score it can, as fast as it can. The eater's sensory input includes the contents of nearby cells, its current score, its color, and number of moves taken so far. Figure 4 also contains a graphical depiction of the input available to an eater (although it is represented completely symbolically for the eater).

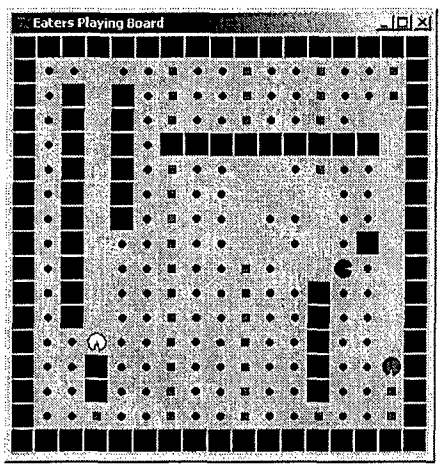

Figure 3: The Eaters Domain

Among the merits of Eaters as our first environment is that it is a hostile environment for episodic memory research. This results from three properties. First, the same features are repeated over and over again, leading to many episodes sharing the same features. Second, the majority of the eater's sensory input changes completely each time it takes a step, greatly reducing the advantages of the interval-based approach. Third, the relatively fast pace of the task requires that many episodic

memories be recorded. However, the simple nature of the domain did make it easy to quantitatively measure the performance of a given agent.

To test the episodic memory system, we created an eater with an episodic memory. This eater does not know the relative value of the objects it senses, nor does it have a model of how its actions affect its world. Our goal is for it to use its episodic memory in place of that knowledge to aid in selecting its action. For each of the possible directions, it creates a memory cue composed of its current sensory input and the proposed direction of travel. The memory retrieved by the pilot implementation represented what happened as a result of the situation described by the memory cue. This new memory includes the score it had received for its last action. This change in score was used to provide an evaluation to a proposed action. The agent then selects the action with the highest evaluation.

There is no guarantee that an appropriate prior episode will be retrieved – that will depend on the set of recorded episodes and the algorithm used for retrieving episodes from memory. If no prior memory is found the agent must still assign a default evaluation to the action so that it can compare the action to the other possible actions. If the default value is greater than bonus food, the eater will be biased to explore new actions, which in turn would build up its episodic memory. If a low value is used, the eater will avoid the unknown.

With the current implementation, we began assessing the eaters agents based upon how accurately they evaluated their available actions. Figure 4 shows a typical result for the current implementation. As the graph depicts, agents using the current version of Soar-EM quickly learn how to improve their behavior in the Eaters environment. After 2000 actions in the world, the agent is predicting the immediate results of its actions with about 90% accuracy. The graph also shows that the agent with instance-based retrieval episodic performs slightly better than the agent with the interval-based retrieval. Both of these result sets are the average of five different agents.

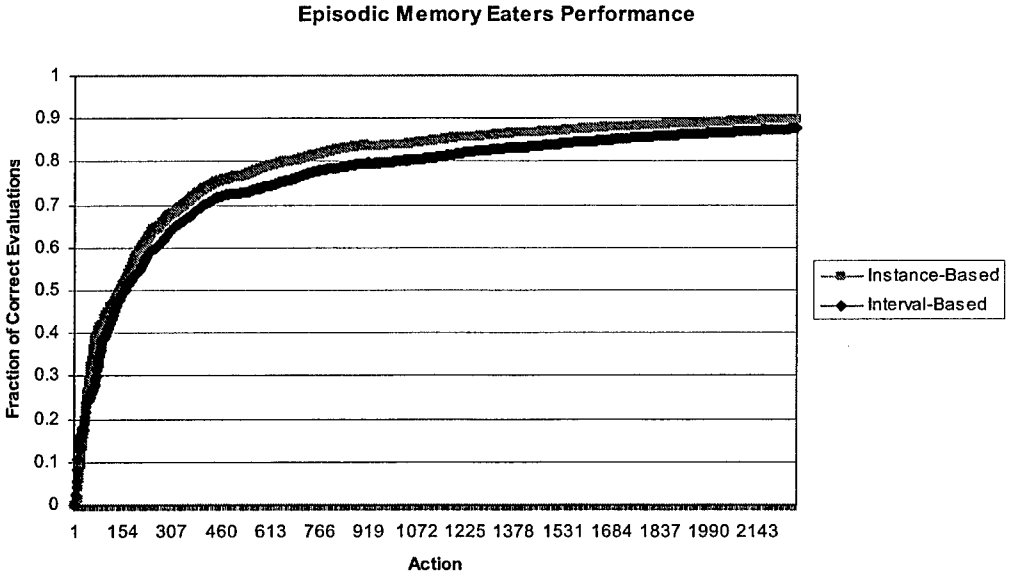

Figure 4: Episodic Memory Eater Performance

Error! Reference source not found. depicts the growth of the additional processing time required for the retrieval operation (by far the most expensive operation).

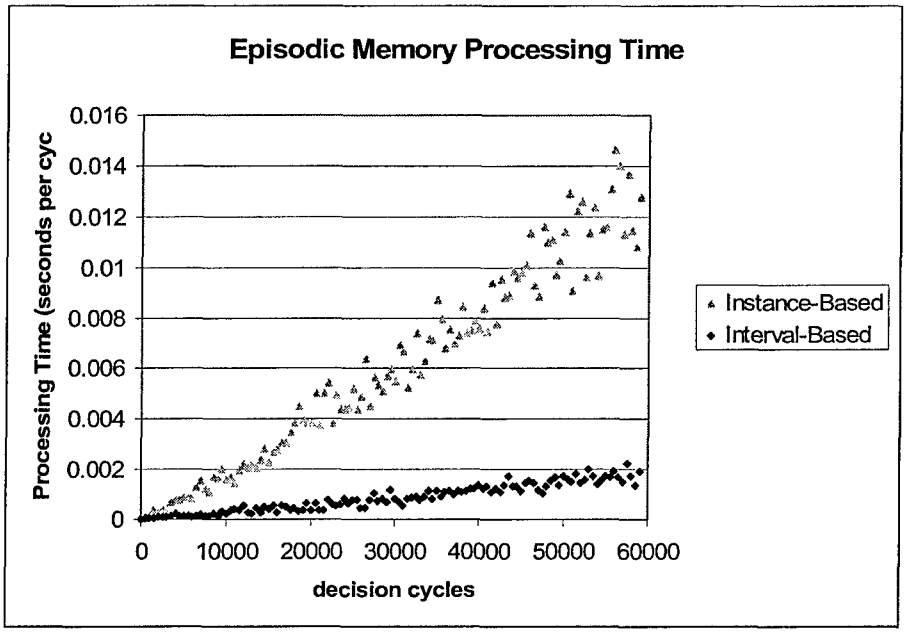

Figure 6: Episodic Memory Eater Processing Time

Figure 7 shows the amount of memory used by the episodic memory system over time. As expected, there is continuous growth over time. However, the instance-based data structures require nearly four times the memory used by the interval-based implementation.

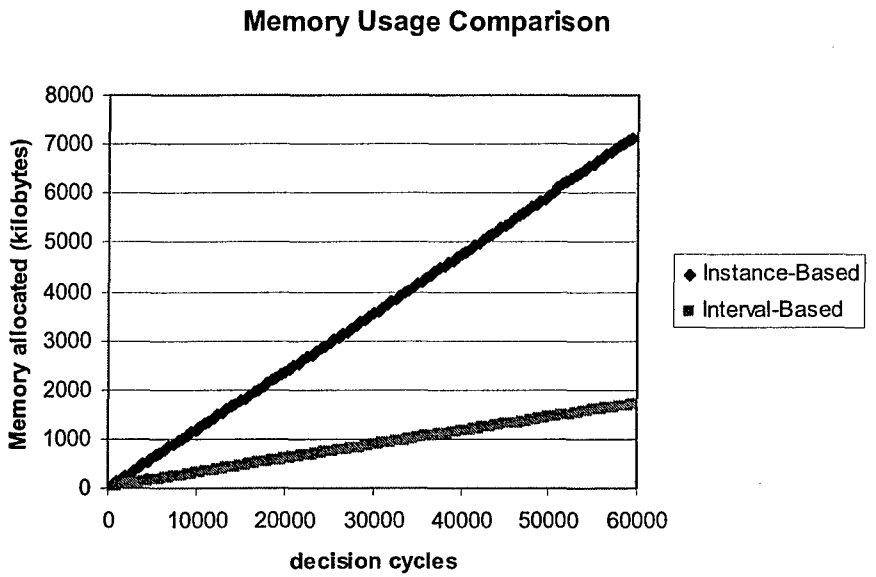

Figure 7: Memory Usage for the Current Episodic Memory Implementation

To reiterate, all these experiments have only been performed on a single domain. Clearly there are trade offs between our instance-based and interval-based approaches. It is our intent to continue to investigation and compare these two algorithms as new environments and algorithm refinements

become available. Eaters has proven to be a valuable test bed for our research, but we must move beyond it to other domains if we want to demonstrate the general usefulness of episodic memory. Our future work in episodic memory is to apply this to a more complex domain.

REINFORCEMENT LEARNING

Reinforcement learning captures statistical regularities of the effect of the entity's actions on the states of the environment as it continually tries to refine its predictions of the future. It learns the expected outcomes of actions that can be used in future decision making. However, it requires multiple related training examples to converge on the correct knowledge and it mostly learns propositional representations for a specific reward function, greatly restricting the generality of the learned knowledge and its usefulness for an entity with dynamic, hierarchical tasks. Our approach to integrating reinforcement learning into Soar starts with the recent integration of probabilistic decision making into Soar. Soar's new decision scheme has two levels of filters, both driven by preferences created by rules matching the current situation. The preferences generated by the first level remove possible choices that are dominated by other choices. If more than one choice survives the first level, the preferences of the second level produce ratings of the expected value for the remaining choices. The ratings for a given choice are summed, and a random selection is made among the choices, biased by the ratings. Soar is distinguished from other reinforcement learning systems because in Soar the computation of the expected value is determined dynamically based on the ratings generated by rule firings, in contrast to the typical approach where a specific value is associated with a state (or set of states) action pair. In those systems, the learner is continually trying to refine the one "true" value for that state-action pair. In contrast, Soar continually learns new rules where each rule captures the value available when it was learned. Each rule defines a set of states that should receive the rating, some rules being more general than others, and the goal of learning is to continually refine the final computation of the rating by learning rules that correct errors in the predicted reward. Moreover, since the rules can include tests for which tasks are being attempted, different sets of rules will be relevant for different tasks, providing a flexible mechanism for supporting multiple tasks with different reward functions.

As stated above, the learning system automatically builds rules that attempt to correct the predicted value (called Q-value) of the selected operator given the results it creates. More formally, the current RL-Soar system uses the following Q-learning update rule:

$$Q(s_t,a_t) = Q(s_t,a_t) + \underline{\alpha(R(s_t,a_t) + \lambda \max_a Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t))}$$

During execution, Q values are updated by inserting new preference rules that carry a numeric value equal to the underlined portion of the above equation. If an operator is selected on the t^{th} decision cycle, then a new preference for that operator will be built during the (t+1) decision.

Inherent to Soar is its ability to automatically generate substates in response to impasses in its problem solving. RL has been successfully applied to systems with hierarchical goals and we have extended our implementation so that it works across Soar's hierarchical structure.

In our implementation, there is an independent reward link on each state in the state hierarchy. Environmental rewards are not distributed automatically to all levels of the hierarchy - each level in Soar-RL only gets rewards that are explicitly given to it. One exception is that for tie impasses when there is insufficient knowledge to choose between options. The purpose of a tie impasse is to reduce the number of candidate operators to one, so we can give a task independent reward for changing the number of tied operators.

We have an implementation that has been tested on a variety of simple domains. There are many research issues before us, with the most important being determining the conditions for the rules and dealing effectively with incomplete state information. The conditions should reflect which aspects of the state are relevant to computing that specific value. If all aspects of the current situation are included, the rules are extremely specific and will only fire in exactly the same situation. We plan to investigate different methods for selecting conditions (initially we will investigate techniques that are based on the most active elements) and for learning with incomplete state information (initially we will investigate techniques based on eligibility-traces).

SEMANTIC LEARNING

This is the least developed of our research initiatives. It deals with learning facts about the world, which on the surface seems the easiest, just store away enduring properties of objects and relationship among objects, as well as properties of and relationships between object classes. However, there are many questions. Which facts should be stored, when should a fact be retrieved, using which cues, what should be done to "update" facts as the world changes, how can this be implemented efficiently, and how is this all done without requiring explicit reasoning by the entity? Our research approach will be the same as above where we decompose the problem of semantic learning into its functional components. The following are the phases of semantic memory according to the general framework presented in our description of episodic memory.

- **Encoding**
  - **Encoding initiation**
    This decision is about how semantic memory encoding is initiated. The options here are deliberate initiation or automatic initiation. 'Deliberate' initiation means encoding is controlled by task specific knowledge encoded in rules. 'Automatic' initiation means encoding is initiated based on general task independent information. The options include: initiate encoding every decision cycle, only when the element is being removed, or to be triggered by some general features such as when there is significant changes in working memory.
  - **Target determination**
    This decision is about what are the structures need to be saved into semantic memory. There are again the options of deliberate determination and automatic determination. Deliberate target determination will be controlled by rules. For automatic determination, there will be several options, such as picking the working memory structure being frequently tested by rules, or picking the one with certain connectivity features, or picking all working memory elements.
  - **Knowledge integration**
    This decision is about how the newly added knowledge is to be integrated with existing knowledge. It's not likely that all declarative chunks are independent structures. Actually, similar structures are likely to combine with each other, which may result in effects such as wider associations and saving of storage space. One simple form of knowledge integration is that when identical structures are repeatedly encoded, they should be merged together.

- **Storage**
  - **Storage structure**
    The structure of semantic memory storage, or the representation, depends on the architecture upon which it is built. The structure must support the other phases of the memory system. One possible design is a graph structure similar to working memory structure, with

additional meta information. Neural network representation is also an option with many desirable properties, but currently, two major problems are computation and connection to abstract symbols that can be used by the symbolic system.

- **Storage dynamics**
  How semantic knowledge change over time. This may include merging or cross indexing with other memories, or removal from the semantic store. The dynamics may result in many interesting phenomena such as forgetting, associative learning and concept formation, *etc.*

- **Retrieval**
  - **Retrieval initiation**
    Acquired knowledge that is relevant to current situation are put into working memory to assist reasoning. Similar to the encoding situation, retrieval can be triggered either deliberately by rule or automatically by task independent mechanisms.
  - **Cue determination**
    Cue is like the stimulus that triggers a response. For semantic memory retrieval, cue is the structure used to retrieve relevant knowledge, which should 'match' the cue according to certain criteria, for example, equality. Cue determination should be task-dependent. It's natural to couple cue determination with retrieval initiation.
  - **Cue specification**
    What is the language to specify the retrieval cue. What should be the expressiveness? Should it support variable, negotiation, disjunctions and other relations? For example, can you say something like retrieve a creature that has the same number of wings as legs (variable and relation), either has feather or fur (disjunction), but not a bird (negation). There will be tradeoffs related to this expressiveness which will be discussed later.
  - **Retrieval algorithm**
    What is the actual algorithm to perform retrieval. The algorithm will be constrained by both the storage and the cue structures. The trade-off between computational complexity and accuracy will be the focus here.
  - **Retrieved result representation**
    How will the retrieved information be represented in working memory. One possible way with minimum invasion is to represent the retrieved information as working memory structures under a specific 'link' (such as that for input and out link).
  - **Retrieval meta-data** (interface)
    What is the extra information needed to be returned from semantic memory? For example, if no matches are found, a failure status needs to be returned; if succeeded, the retrieval confidence value for best partial match might be useful.

- **Use**
  How the retrieve knowledge is used will depend on task specific knowledge encoded by rules.

The most general criteria for evaluating a design implementation include the following. The first four are task-independent and the fifth is task-dependent. They reflect the various aspects of the requirements for a good design.

- **Efficiency:** The computational complexity to find the target from semantic memory for a given cue should be constant or near constant time, or otherwise at least bounded. Retrieval time should scale up well to be feasible for large applications.

- **Flexibility:** The cue specification needs to be flexible enough, so that a sufficiently expressive specification language is required. The natural tradeoff is that more expressive specification results in more expensive computation. For example, disjunctions will increase the flexibility of query, but also easily enlarge the searching space for the matched candidates.
- **Accuracy:** If it is not feasible to always find the global optimum of the best partial match and approximation has to be done, then to what degree should accuracy be compromised? Accuracy is also
- **Ease of use:** If automatic encoding is used instead of deliberate encoding, one impact is that there is no need to program extra rules to control semantic knowledge encoding.
- **Efficacy:** Evaluation of efficacy is task-dependent. In many situations, it is not easy to discovery general criteria across tasks for different domains, thus the evaluation need to be done task by task.

Exploring design space

The design framework helps identify critical decision points and make it explicit to explore the complex design space in a systemic way. In order to find optimal designs, the design options will be evaluated either by both general criteria and representative tasks.

Current implementation

One general principle of the design is to be minimum intrusive. The added semantic memory should be well integrated without affecting the other mechanisms, although in the long term, there can be more radical modifications. Following is the current implementation described according to the general design framework.

- **Encoding**
  - **Encoding initiation**
    Both deliberate and automatic encoding options have been implemented. Deliberate initiation is via a special working memory structure, the 'save' link. If automatic encoding is turned on, it will initiate encoding at the output phase for each decision cycle.
  - **Target determination**
    Deliberate target determination is also mediated via the save link. Task knowledge will help identify what structures need to be save and put them under the save link. For automatic determination, all working memory elements are saved, except for the cue structure itself.
  - **Knowledge integration**
    Currently, the only integration operation is merging identical declarative *chunks*. A declarative chunk is defined as a set of working memory elements with the same identifier, which describe some coherent object or concept. Therefore, an identifier uniquely identifies a declarative chunk. Two declarative chunks with different identifiers are considered identical if they have exactly the same set of attribute-value pairs and will be merged together. It is a common situation that identical structures are created at different decision cycles or under different context. On the other hand, different attributes for the same chunk could be saved into semantic memory at different cycles. Figure 8 shows an example, step by step, about how semantic knowledge is encoded.

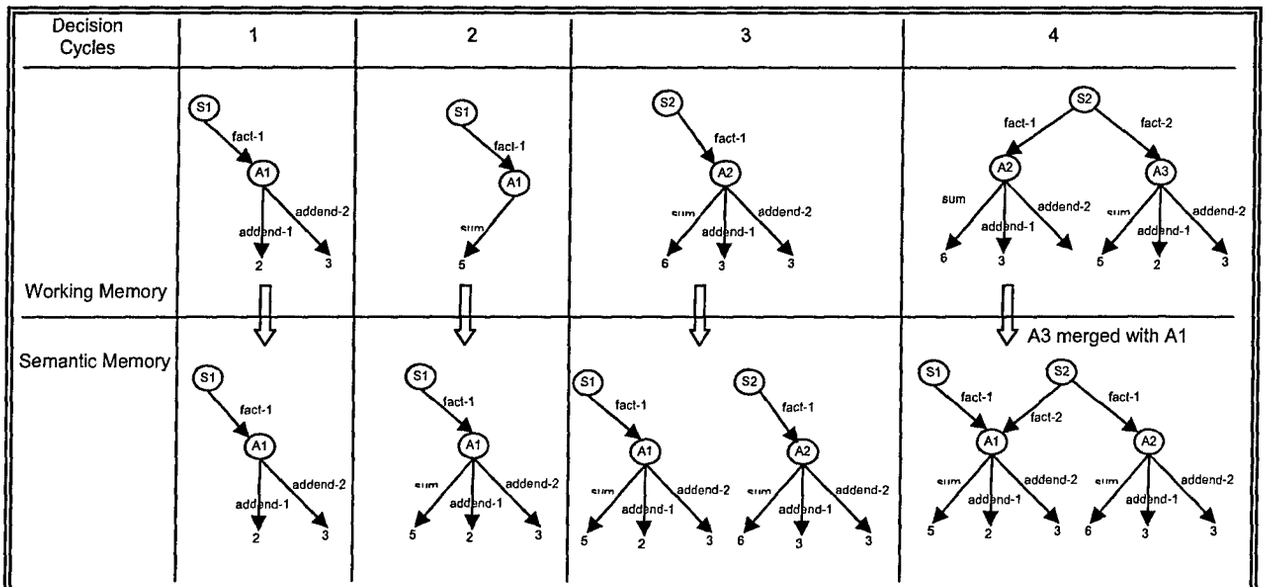

Figure 8: Encoding

At decision cycle 1, fact-1 which contains two attributes is deposited into semantic memory. At decision cycle 2, fact-1 has a new attribute and the previous attributes are removed from working memory, but in long term semantic memory, all three attributes are saved under the same declarative chunk with identifier A1, which is 2+3=5. At decision cycle 3, a sub-state S2 is created, which contains another fact-1: 3+3=6, this new structure is also deposited into semantic memory. At decision cycle 4, a new fact, fact-2 which is 2+3=5 is added to working memory under S2. The new chunk A3 is added to working memory but it is identical ti chunk A1, so they merged together, and the value of fact-2 under S2 becomes A1 instead A3.

- **Storage**
  - **Storage structure**
    At the conceptual level, the storage structures are declarative chunks. Each declarative chunk consists of triples of identifier, attribute and value just as working memory elements. Identifier is unique for each declarative chunk. There are data structures facilitating operations performed on semantic knowledge (Figure 9).

  - **Storage dynamics** (internal)
    For current implementation, the only dynamics is to add new long term elements into semantic memory. There is no removal and forgetting yet, nor is there automatic association of similar chunks. Storage dynamics share similar routine with knowledge integration of the encoding phase. The main difference is the latter is triggered at encoding time, the former is anytime.

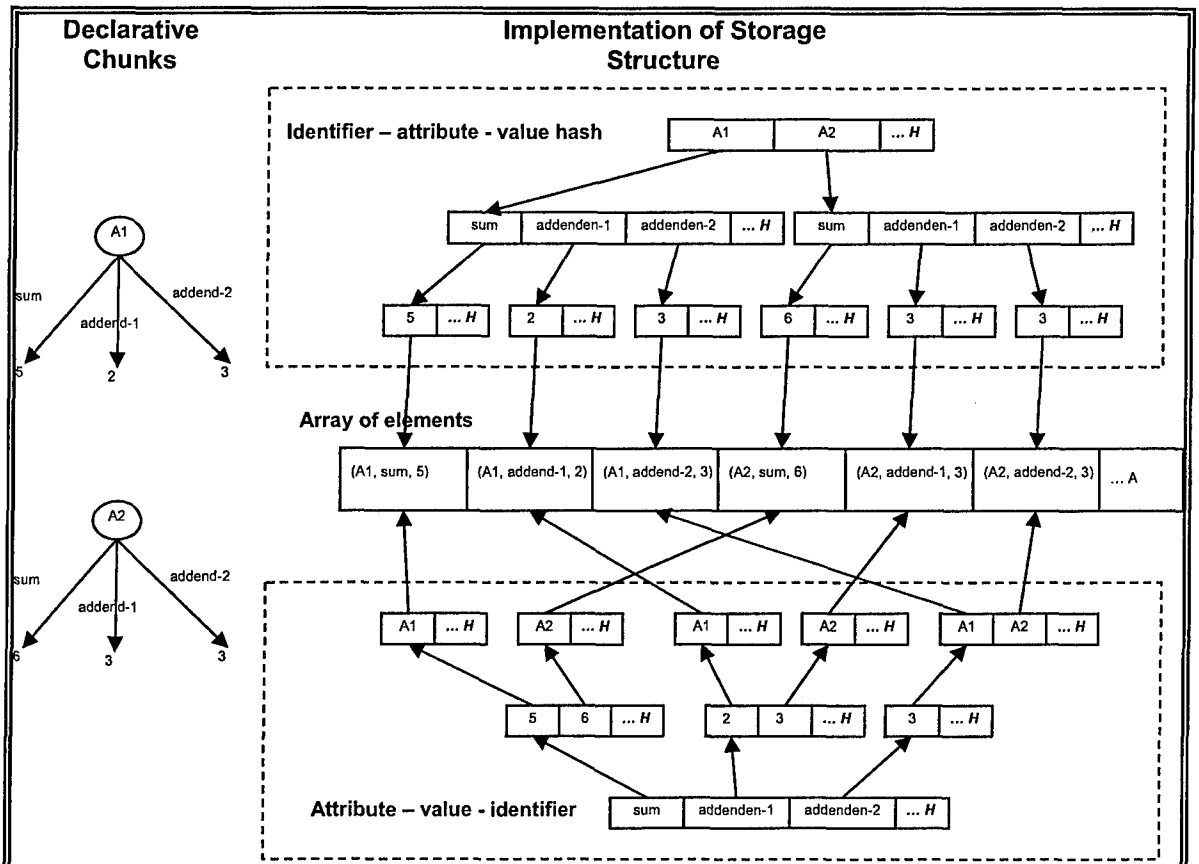

Figure 9: Storage

The figure shows semantic memory storage structure for the situation when it contains only two chunks for addition facts: 2+3=5 and 3+3=6. There are two hierarchical hashing structures to facilitate operations, such as *retrieval* and *merging* in semantic memory. '...A' denotes array data structure, and '...H' denotes hash data structure.

Attribute-value-identifier hash will facilitate attribute-value based matching operations. For the first level hash, the keys are attribute names, and the value for a unique key is the second level hash. The second level hash uses attribute values under the same attribute name as the key and the third level hash as the value. And the third level hash uses the identifier having the same attribute-value pair as the key and the pointer to the thus far unique identified triplet structure which is stored in an array. The triple element may contain extra information such as the reference history and other data structures for the intended purposes.

Similarly, the identifier-attribute-value hash will facilitate retrieving the attributes being queried based on the matched identifier.

- **Retrieval**
  - **Retrieval initiation**
    Currently, retrieval initiation is triggered deliberately via a special working memory structure, the 'cue' link.
  - **Cue determination**
    Cue determination is coupled with retrieval initiation via the 'cue' link. Cue structure is determined by task specific knowledge which is encoded as rules.
  - **Cue specification**
    Currently, the cue specification language support one level retrieval with variable, but doesn't support negations, disjunctions and relations other than equality.
  - **Retrieval algorithm**
    Current implementation of retrieval algorithm is a complete search algorithm that finds exact matches. If multiple matches are found, an arbitrary selection is made by the algorithm. If no match is found, a failure chunk is returned.

- **Retrieved result representation**
  Retrieved information is put under a special working memory structure, the 'retrieved' link.
- **Retrieval meta-data**
  In current implementation, the only meta-data is the failure chunk to indicate the situation that no match can be found from the semantic memory.

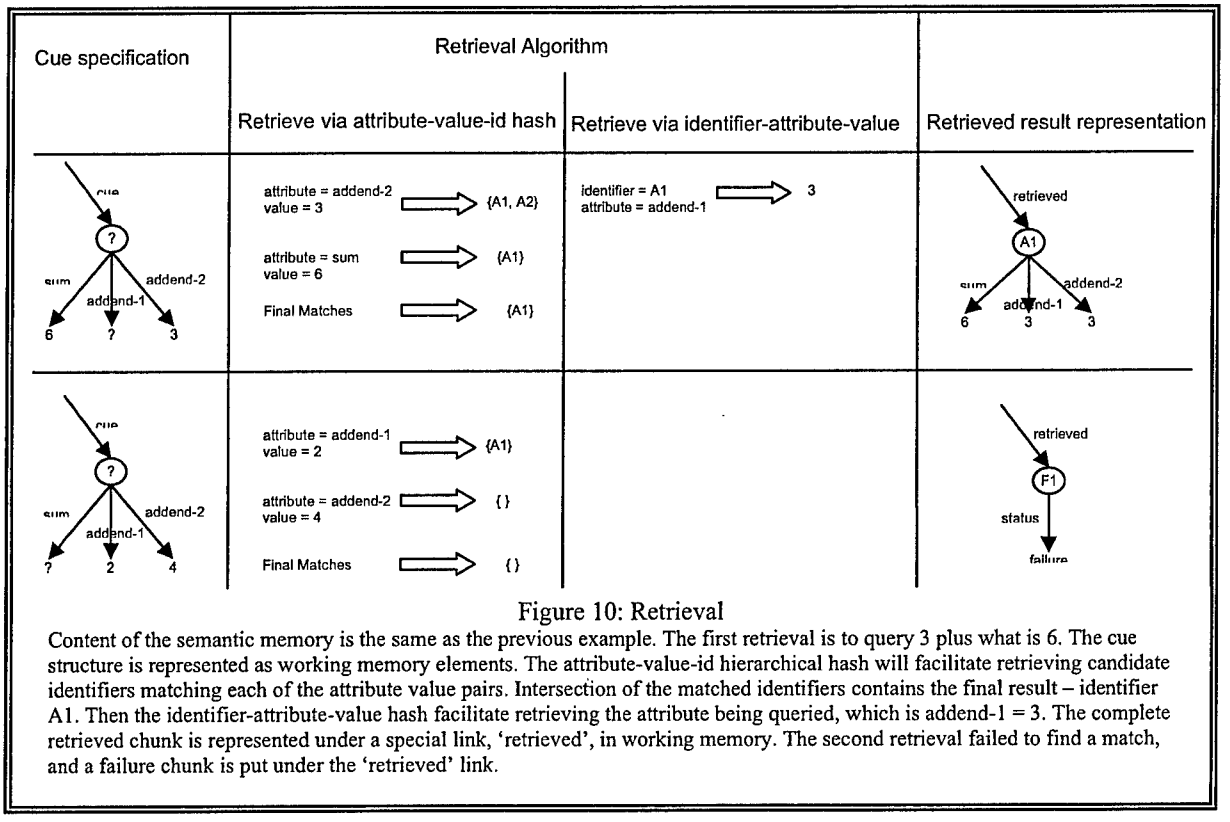

Figure 10: Retrieval

Content of the semantic memory is the same as the previous example. The first retrieval is to query 3 plus what is 6. The cue structure is represented as working memory elements. The attribute-value-id hierarchical hash will facilitate retrieving candidate identifiers matching each of the attribute value pairs. Intersection of the matched identifiers contains the final result – identifier A1. Then the identifier-attribute-value hash facilitate retrieving the attribute being queried, which is addend-1 = 3. The complete retrieved chunk is represented under a special link, 'retrieved', in working memory. The second retrieval failed to find a match, and a failure chunk is put under the 'retrieved' link.

Preliminary Experiments and Analysis

Preliminary experiments on some representative tasks have been performed to test the current implementation.

Associative learning task

An obvious application of semantic learning is to tasks requiring learning associations, such as required in arithmetic. The standard addition procedure used by humans is to process the problem column by column (Figure 11). If the sum for a column is over ten, an extra one is carried to next column. This task demonstrated the desired semantic memory functionality of flexible knowledge representation and the related effects in transfer learning.

To better demonstrate learning, the initial available knowledge contains just counting facts: 0 is followed by 1, which followed by 2, *etc.*... up to 9. The agent doesn't have knowledge about addition results, such as 2+3=5, but it knows how to do a parallel counting from 0 to 3 and 2 to 5, and then obtain 5 as the final result (Figure 6-b). It also knows to carry 1 to next column if it counts over 9 (then continue from 0) when doing addition, or borrow from next column if it counts below zero (then continue from 9) when doing subtraction. The counting facts are encoded as working memory structure with next pointers connecting the 10 digits, from 0 to 9 (Figure 6-b). Other

knowledge is encoded as production rules. Random addition/subtraction problem are generated in working memory to be solved. Using the semantic learning component, the system must initially use a primitive counting procedure to do addition (or subtraction). Over time, the addition facts are built up in semantic memory so that instead of counting, and memory retrieval can be used. Note, that an addition fact can be used for subtraction and vice versa. As expected, this leads to a speed up in performance over time.

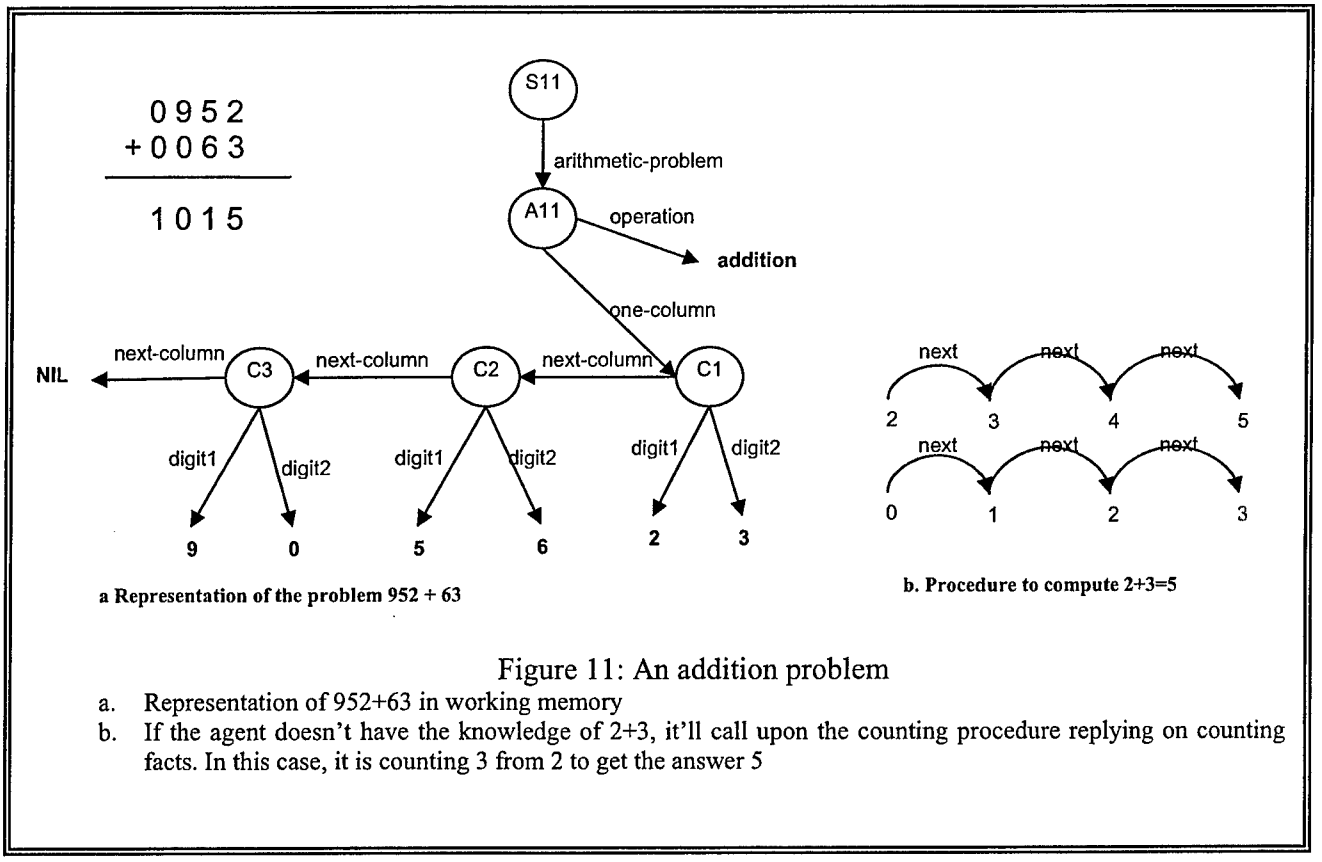

Figure 11: An addition problem

a. Representation of 952+63 in working memory
b. If the agent doesn't have the knowledge of 2+3, it'll call upon the counting procedure replying on counting facts. In this case, it is counting 3 from 2 to get the answer 5

Transfer learning

Chunking and semantic learning are complementary. Since retrieval is a more expensive operation than firing a rule, learning rules that replace retrievals is practically very useful. As can be seen from this example, if retrieval takes place in a sub-state, the retrieval process itself can be compiled into a rule by chunking. Intuitively, the purpose of chunking is to speed up execution via practicing, and semantic learning is to learn flexible knowledge structure potentially transferable to different procedures without practicing with the exact situation. The following result demonstrates the different aspects of learning in a simple example (Figure 12).

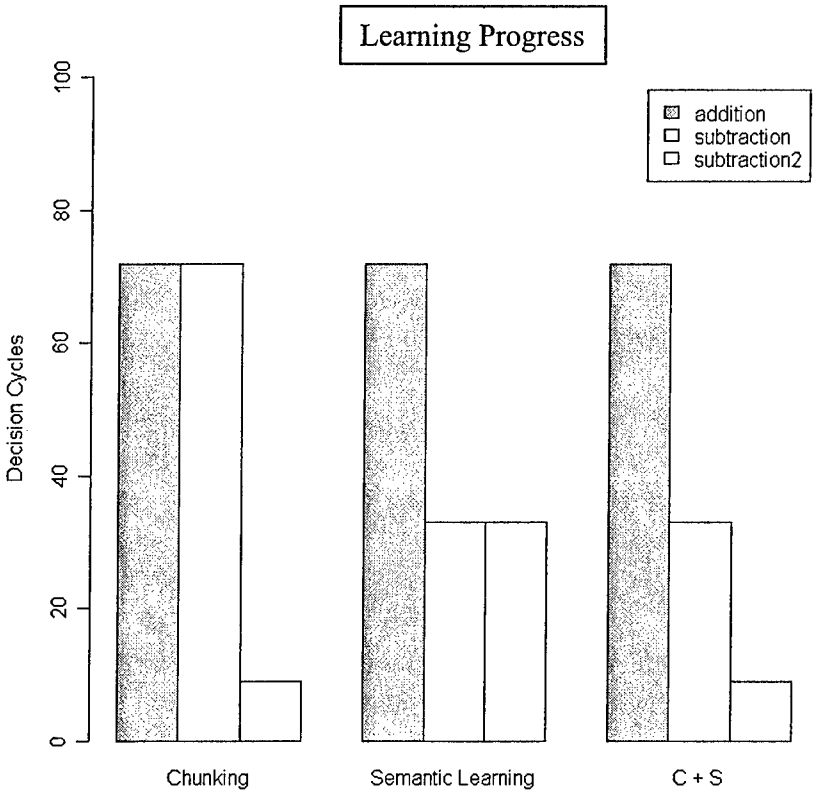

Figure 12: Semantic learning helps achieve transfer learning

The addition problem in this experiment is: 234 + 345 = 579. The subtraction problem is: 579 - 345 = 234.
The agent did the addition problem once, and then did the subtraction problem twice.
Chunking doesn't have transfer learning effects as the learned addition rule is not applicable to subtraction. But chunking will greatly speed up the problem solving by practicing with the same problem
Semantic learning is able to transfer knowledge from addition to subtraction.
Having chunking and semantic learning work together (C + S), it will benefit from both aspects of learning.
Decision cycles: Chunking: 72, 72, 9; Semantic Learning: 72, 33, 33; C + S: 72, 33, 9

In real world problems, the agent doesn't have complete knowledge or exact model of the task environment, so that uncertainty emerges. Different from the exact production rule matching mechanism, semantic memory provides *best partial match* to better handle uncertainty. Assumptions about the environment need to be made. The following are three principles about best partial match, which are based on Anderson's *rational analysis*.

Future

As mentioned earlier, this work has continued under the BICA program. We are extending these learning mechanism, studying their integration, and also developing a computational theory of emotion.